

Modern Game AI Algorithms

Assignment 1 - PCG in Minecraft

Kevin Bretz

Leiden University

March 2, 2025

1 Introduction

Procedural Content Generation (PCG) has become an essential tool in modern game development, allowing for the creation of dynamic and unpredictable environments. In sandbox games like Minecraft, PCG is especially useful for enhancing replayability and providing unique player experiences. This report describes the design and implementation of a procedural house generator for Minecraft. The goal was to construct a house that adapts to its environment and will successfully be built regardless of terrain.

The system begins by locating a viable starting point and recursively searches for additional room spots. Each room is randomly sized, and multi-level buildings are constructed with diverse palettes, roof styles, and entrances. Additional features like fences, foliage, and animal spawns enhance the realism of the generated structures.

A major challenge was handling diverse terrains. An initial randomness-based search struggled with complex biomes, leading to the implementation of a more exhaustive search algorithm. This improved reliability but increased processing time—a worthwhile trade-off for consistent results.

This report outlines the generator's pipeline, from the search algorithm to structure assembly, highlighting key design decisions and potential improvements for future iterations.

2 The Pipeline

Here a short overview of the individual steps the program goes through.

1. Search for viable spot for first room
2. Recursively search for viable spot in immediate surrounding
3. When no more room spot can be found, build the ground floor

4. Add floors to each room, build an entrance (if possible)
5. Add windows and roofs
6. Build a fence, add foliage, spawn animals

2.1 Search

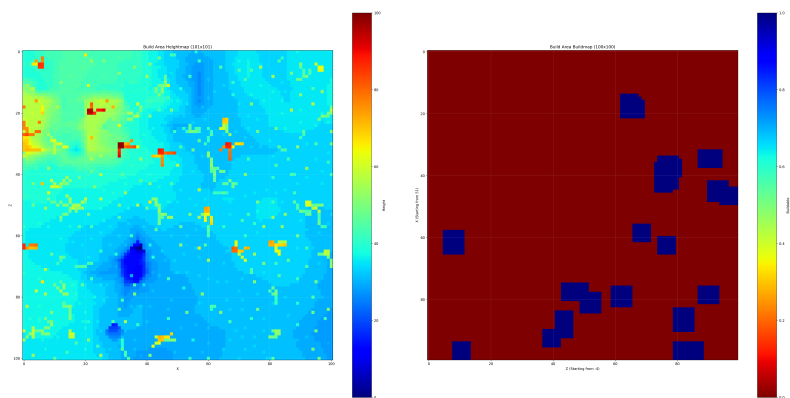
For locating a spot for the first room I originally used a method that implemented randomness into the search algorithm. Although this was convenient for testing, as it reduced the processing time, it often was unable to find a suitable spot if the terrain was particularly difficult. This was usually the case for jungles, as well as some mountains.

In later stages of development, I adapted this to an algorithm which searches through the entire build area and retrieves all possible room locations for all possible room dimensions. This eliminated the problem of not finding a suitable building spot in virtually every biome except the ocean. However, on flatter terrain, it drastically increased the processing time, as there are exponentially more options. Due to loading time not being of importance for this assignment, this is a worthwhile payoff, although quite annoying to work with.

The criteria for a spot to be suitable for a room is that an area of at least 6x6 must be completely flat. It can not build on water, lava, or ice, and ignores leaves. The room dimensions are chosen randomly, and each wall can be anywhere in between 6 to 10 blocks wide, as well as tall. Room width, length and height are independent of each other.

Originally, I wanted to write a code that searches for the best spot for a room based on the gradient of the environment. Just for simplicity reasons I started out with this basic algorithm, planning to improve on it. However, I was pleasantly surprised to find that the basic algorithm, even in it's previous, more rng-dependent iteration, almost always found a suitable spot. With the before mentioned improvement, it basically guarantees a structure will spawn unless the build area is defined over an ocean.

The buildmap visualizes a 2D plot of the entire build area. Blue squares are buildable spots, red squares the opposite. Due to the nature of the search algorithm, a large part of development was spent testing on a superflat world, which is a world in minecraft where every block has the same height, it is completely flat.

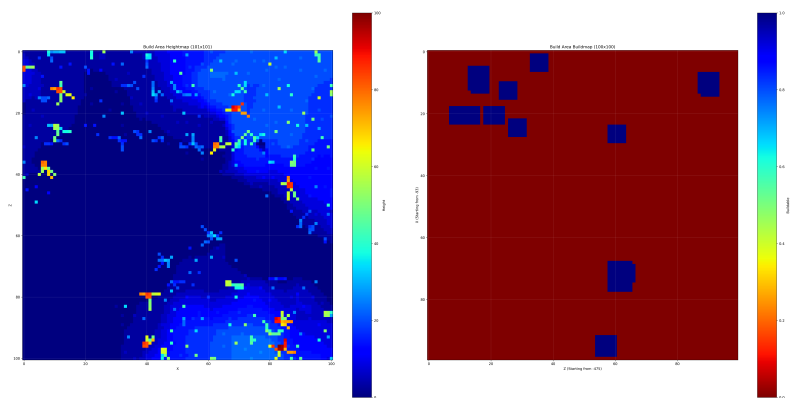


(a) Heightmap of jungle example 1

(b) Buildmap of jungle example 1



(c) Jungle example 1

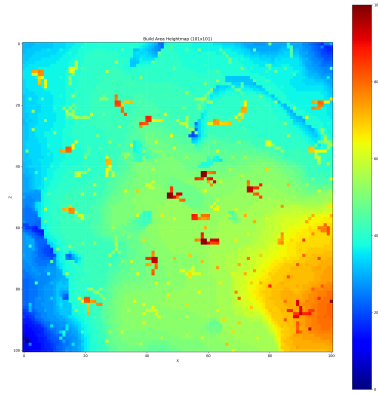


(d) Heightmap of jungle example 1

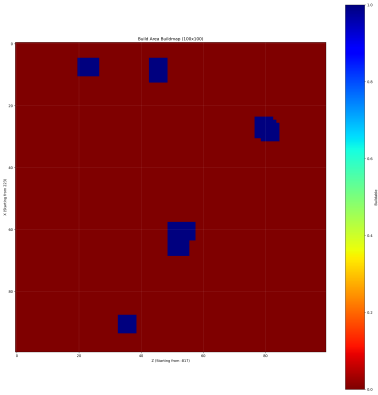
(e) Buildmap of jungle example 2



(f) Jungle example 2



(g) Heightmap of jungle example 3



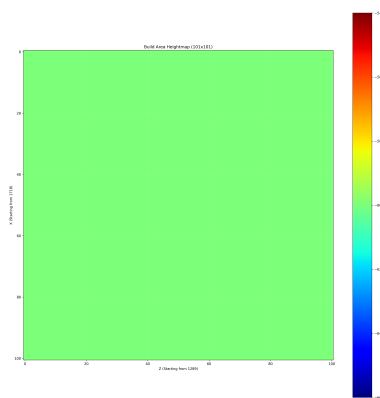
(h) Buildmap of jungle example 3



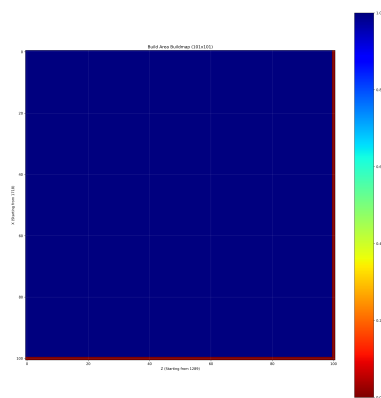
(i) Jungle example 3



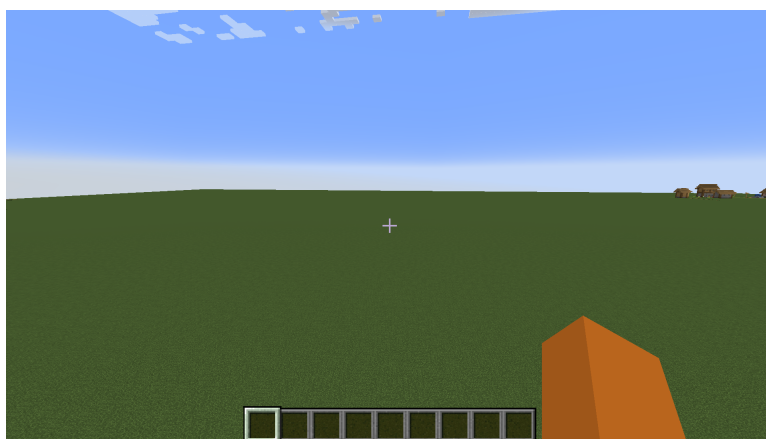
(j) Jungle house



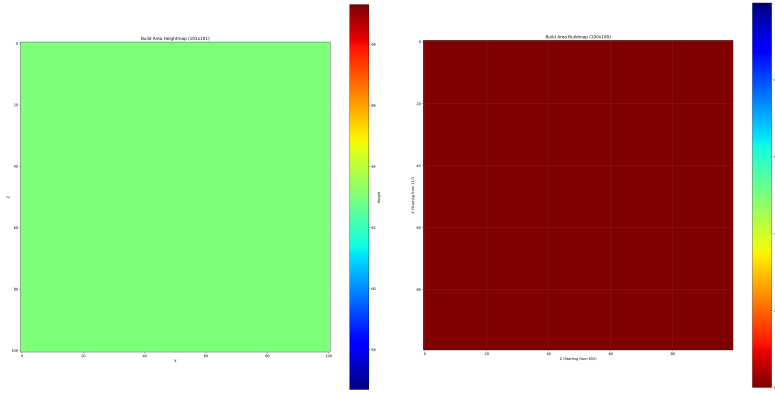
(k) Heightmap of superflat world



(l) Buildmap of superflat world



(m) Superflat world



(n) Heightmap of ocean

(o) Buildmap of ocean



(p) Ocean

2.2 Expand

Next, the corner coordinates of the room are saved, as well as its dimensions. Using this, a new search function, which uses the same basic idea as the previous one but implements more randomness and a chance that the room search can fail even if there are viable spots, searches if there is a suitable spot in the immediate surroundings of the first room. If this is the case, that spot is again saved, and the same search algorithm reapplied to it, until no new viable room spot is detected. Applied to a superflat world, this allows for potentially endless expansion. (I recommend you try this out yourself!)

Figure 1: Single structure generated on superflat terrain



2.3 Build

After recursively searching for room spots, the algorithm goes through all saved rooms and constructs them, adding a random number of levels to each room, up to a maximum of 4 floors. A door is placed in the wall that intersects 2 rooms. When running the code, a specific palette is defined for the house, and all rooms will be built using materials contained in this palette. I created a total of 5, hopefully aesthetically pleasing, palettes. Additionally, the roof and lighting design of each room is chosen individually, and the color of the windows is even defined per level to add more randomness. If a fitting spot can be allocated, a stylish entrance is built. Finally, a fence is built around the perimeter of all rooms, random foliage is placed within the garden, and some random animals are spawned.



(a) Cozy palette



(b) Cottage palette



(c) Elegant palette



(d) Rustic palette



(e) Modern palette

3 Example Images



(f) Roofs designs, snowy terrain



(g) Forest Terrain



(h) Desert terrain



(i) Indoors

4 Possible improvements

Besides obvious improvements such as more palettes and more individual designs for entrances, roofs, lighting, etc., there were many ideas and concepts I began implementing, but then scratched because of time constraints.

First of all, and quite obviously, fitting interior decoration and furniture should be added to the rooms. I initially worked on implementing a function that builds a spiral staircase. However, after losing over a day of time on it, I eventually gave up and resorted to adding ladders. An improved version should have fancier and more realistic ways of traversing levels than ladders, and interconnections between the different levels as well. The search algorithm could also be improved. Although with my method the chances of not finding a suitable build area are extremely low, adding gradient handling would allow for the house to expand regardless of steepness level. Finally, although I don't mention it a lot because it isn't listed as relevant for the assignment, the code could be heavily optimized and performance drastically improved.

5 Conclusion

The procedural generation system described in this paper successfully creates multi-room houses in Minecraft that feel organic and fit within the game's aesthetic. By combining a thorough search algorithm with randomized architectural features, the system reliably finds suitable building spots across different terrains and produces structures with a good amount of variation. Although the initial randomness-based search worked quickly, it struggled with complex environments like jungles. Switching to an exhaustive search solved this issue, making it almost guaranteed that a structure will be generated unless the build area is over an ocean.

The recursive room expansion approach allows for dynamic and unpredictable house layouts, and the randomized palettes, roof styles, and exterior details add personality to each generated structure. There are still clear areas for improvement. For example, adding proper staircases instead of simple ladders would make the houses feel more realistic. Likewise, implementing interior decoration could make the structures feel more lived-in and complete.

Future improvements could also focus on optimizing performance and refining the search algorithm to handle uneven terrain better. Despite these limitations, the current system achieves its primary goal: creating varied and visually interesting Minecraft houses with minimal manual intervention. This project highlights how procedural generation can produce meaningful content while balancing complexity, randomness, and design constraints.

6 References

1. van der Staaij, A. (n.d.). GDPC: Minecraft datapack creation framework [Computer software]. GitHub. <https://github.com/avdstaij/gdpc>